USER INTERFACE AND FUNCTION LIBRARY FOR GROUND ROBOT NAVIGATION

By

Micah Smith, B.S.

A Project Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

University of Alaska Fairbanks

May 2017

APPROVED:

Orion Lawlor, Committee Chair
Jon Genetti, Committee Member
Glenn Chappell, Committee Member
Jon Genetti, Chair
Department of Computer Science

User Interface and Function Library for Ground Robot Navigation

Micah Smith

April, 2017

COMMITTEE:

Dr. Orion Lawlor

Dr. Glenn Chappell

Dr. Jon Genetti

Abstract

A web application user interface and function library were developed to enable a user to program a ground robot to navigate autonomously. The user interface includes modules for generating a grid of obstacles from a map image, setting waypoints for a path through the map, and programming a robot in a code editor to navigate autonomously. The algorithm used for navigation is an A* algorithm modified with obstacle padding to accommodate the width of the robot and path smoothing to simplify the paths. The user interface and functions were designed to be simple so that users without technical backgrounds can use them, and by doing so they can engage in the development process of human-centered robots. The navigation functions were successful in finding paths in test configurations, and the performance of the algorithms was fast enough for user interactivity up to a certain limit of grid cell sizes.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Autonomous Navigation in Human-Centered Robotics

The once fantastic idea of robots interacting with humans in our daily lives is beginning to become a reality. Many companies and researchers are designing robots for use in healthcare, education, search and rescue, and daily life situations [1].

Autonomous navigation in spaces with humans is one difficult aspect of human-centered robotics that presents both technical and social challenges. It requires efficient algorithms to navigate complex and changing spaces, but it also requires careful decisions in how the robots should respond to human interactions [2].

End users and experts in fields such as sociology, education, and healthcare without technical backgrounds can be instrumental in addressing these social challenges, but the technology must be made simple and accessible in order for them to play direct roles in developing robot behavior.

Engaging in the development process of human-centered robotics could serve as a bridge for some users to the fields of science and technology [3]. Robots are unique in the field of technology for their ability to perform tasks and interact with humans, and users from other backgrounds may discover novel ways to use them in their own fields.

1.2 Purpose

The purpose of this project was to develop a lightweight frontend user interface for programming autonomous navigation in ground robots that is simple enough for users without a background in programming or engineering.

The user interface was developed on top of an existing web frontend developed by students and faculty at the University of Alaska Fairbanks, RobotMoose [4]. RobotMoose was

designed to be used by students in grades 6-12 to operate robots remotely. RobotMoose provides the ability to drive ground robots using an on-screen wheel or to program robot movement using javascript and a simple function library. But it lacked tools for autonomous navigation such as path planning algorithms and a map representation for storing boundaries and obstacles.

This project aimed to expand the user interface and function library of RobotMoose to allow a user to upload a map image, generate a grid of obstacles from the image, plan a path through the map by setting waypoints, and program the robot using simple lines of code to navigate autonomously between the waypoints.

1.3 RobotMoose Infrastructure

RobotMoose is a web-based application written in Javascript. It has a user interface for operating and programming robot actuators and visualizing sensor data, including an embedded Gruveo video interface for telepresence robots and a WebGL display of a grid and 3D model of an iRobot Create 2 robot [5] to visualize movement when a ground robot is connected. Figure 1 shows the interface for operating a robot. RobotMoose communicates with the robot through a custom web server intermediary called Superstar. Superstar runs a JSON API and also serves as a JSON database to store data for each robot. A RobotMoose Chrome App is normally used as the backend software to send commands and receive sensor data through a serial port to the robot microcontroller [6].

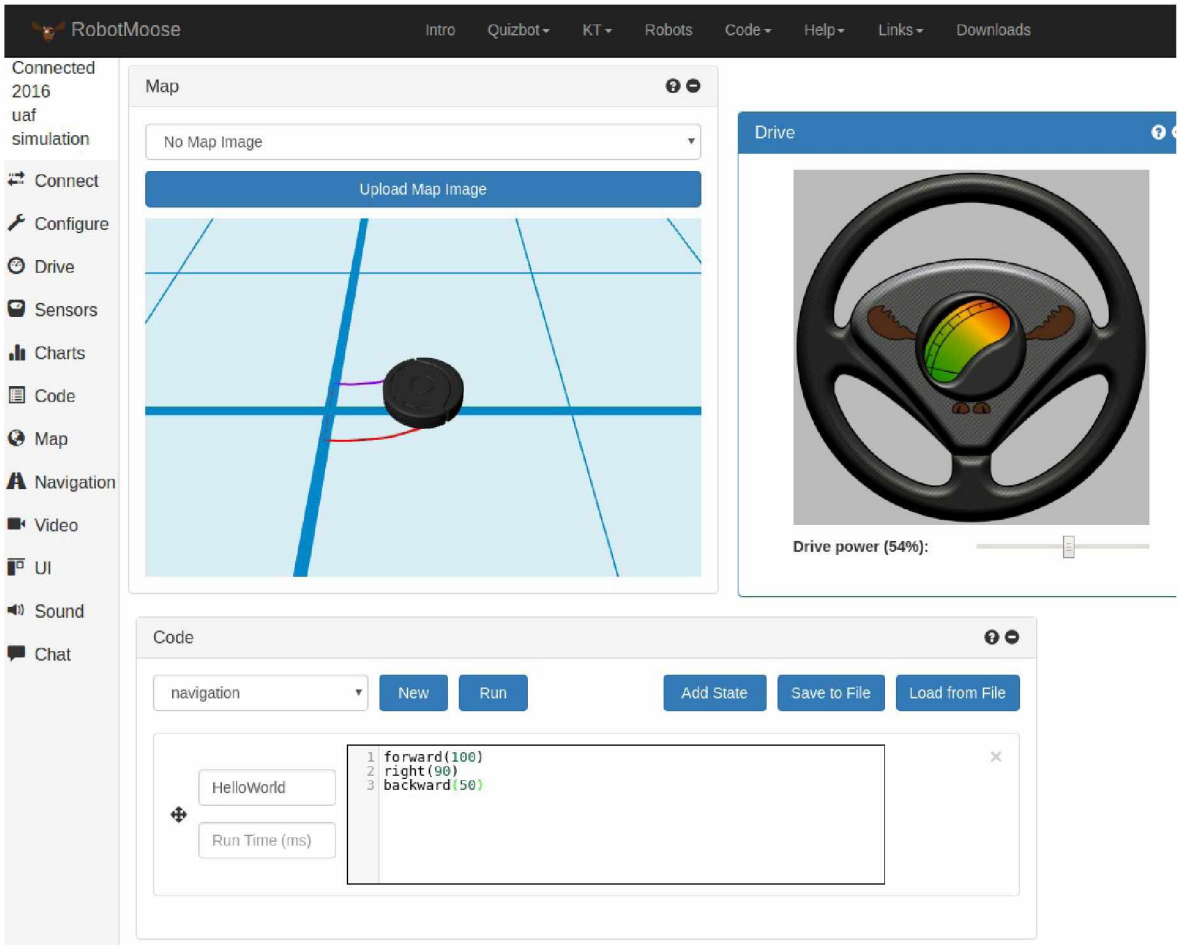

Figure 1: Existing RobotMoose User Interface. Shows the 3D map, code editor, and drive wheel. The links in the side bar are to modules in the user interface. The links in the top navigation bar are to other resources on RobotMoose.com.

2 RELATED WORK

2.1 Robot Operating System

The Robot Operating System (ROS) is a commonly used set of tools and libraries for developing robotic systems [7]. One available package is a navigation stack with algorithms for autonomous navigation [8]. It uses a global planner to find a path through a map of obstacles with Dijkstra's algorithm or A* path planning. After calculating the global path, it uses a local planner to determine velocity commands to send to the robot. One difficulty with ROS is that it can be a complicated process to install the packages and configure a robot to use them, and the software only supports a limited number of Linux-based platforms.

ROS has a frontend user interface package that can visualize the robot and connect to the navigation stack, Rviz. Rviz has most of the functionality desired in this project, such as the ability to plan a global path for autonomous navigation by setting waypoints on a map. But it is a robust package that works with many kinds of robots for many tasks beyond ground robots and navigation. Using it for navigation requires many steps to setup and load in shapes or models for the map, robot, and obstacles. So it may not be accessible for non-expert users without significant previous setup or training. Rviz must be installed on a computer with ROS and does not run in a web browser. Running a simulation requires connecting to a separate simulation package, either from ROS or a third party [9].

ROS has a package with similar functionality to Rviz that runs in a web browser, Wviz [10]. Wviz can connect to a robot with the ROS navigation stack and visualize the map in much the same way as Rviz. But without prior setup Wviz starts as a blank slate, and the user must add elements such as map, robot model, and obstacles. It shares the drawbacks of Rviz that it is overly robust for non-expert users without setup and training, and it has no built-in simulation. Though Wviz runs in a web browser, it also shares the drawback of all ROS

packages that setting up a robot or simulation for operation is a complicated process, and the platforms it supports are limited.

2.2 Applications for developers

A number of companies and researchers have developed other frontend applications that connect to robots running ROS or similar software on the backend. Some of these are designed to be used by other developers, such as V-REP from Coppelia Robotics. V-REP is a robot simulator and development environment supporting a wide variety of development approaches. The developer can control the robot using several scripting languages, a remote API, an ROS node, or other plugins and custom solutions. It has built-in libraries for path planning and physical simulations. While it is designed for simulations, the same remote commands can be copied to a real robot. Like Rviz and Wviz, this software is probably too robust and complex for non-technical users, and it requires time for installation and setup [11].

2.3 Applications for End Users

There are other frontend applications for operating robots designed for non-technical users. These are often simple to use but lacking in the ability to involve the user in robot development. One example is the Oculus Prime robot from Xaxxon [12]. Oculus Prime is a commercial security robot with a frontend web browser interface for driving a robot and setting waypoints for autonomous navigation. The user interface is intuitive and simple. But the application is designed for a specific purpose of making a path for patrolling an environment, so it does not provide any options such as programming or scripting that would involve the user in a development process or decisions about interactions with humans.

There are other robots designed specifically for human interaction, such as the RP-VITA Telemedicine Robot from inTouch Health and iRobot [1]. RP-VITA is a telepresence robot

that allows a doctor or caregiver to interact with a patient remotely. It has a simple user interface for operating the robot and viewing patient data. It uses close-range sensors and mapping technology to detect and avoid obstacles. It also has the capacity for fully autonomous navigation. The user can tell the robot to drive to a location by clicking on a map. While RP-VITA is an excellent model of human-centered robotics, as a commercial healthcare robot it is not designed to be used as a tool for robotics development.

A number of applications are being released for educational purposes, to allow young students to operate robots and learn about robotics, technology, and programming. Tickle is one popular example that allows the user to operate robots such as Sphero and Ollie using drag-and-drop programming [14]. Tickle makes programming robots accessible even to users without technical experience. Users can program the robots to navigate, and they can make decisions about how the robots should respond to various sensor inputs. Tickle can be very helpful for giving users experience with robotics and the development process. But the Tickle application and supported robots are limited in functionality. The application does not support mapping or path finding algorithms, and the robots have limited sensors and actuators for interacting with the environment.

3 APPROACH

3.1 Overview

The RobotMoose frontend was expanded to enable a user to program a robot to navigate autonomously through a space using a map. This process comprises five tasks: 1) Upload a map image. 2) Identify obstacles from the map image. 3) Set the starting location and goal on the map. 4) Find a path to the goal. 5) Drive to the goal.

3.2 User Interface

To enable the user to easily experiment with different maps, robot paths, and programming code, the RobotMoose frontend was expanded to include a built-in ground robot simulation. It simulates a ground robot with wheel base and velocity approximately equal to those of an iRobot Create 2 robot. If a simulation is running, RobotMoose sends commands to the simulation rather than to the Superstar server. When frontend drive commands set the power of the simulated robot's wheels, the simulation starts a drive loop that moves the robot's sensor values incrementally at each iteration. Data that would be stored in the Superstar database for real robots is stored in the browser's local storage for simulations. This data includes robot configuration, user interface configuration, and stored driving programs.

To enable the user to simulate driving a robot in a real space, a button was added above the WebGL display to upload a map image from the local machine. See Figure 2 for the interface. The user sets the scale for the map, and the uploaded image is displayed on the grid in the WebGL screen so the 3D model robot can be driven on it, as shown in Figure 3.

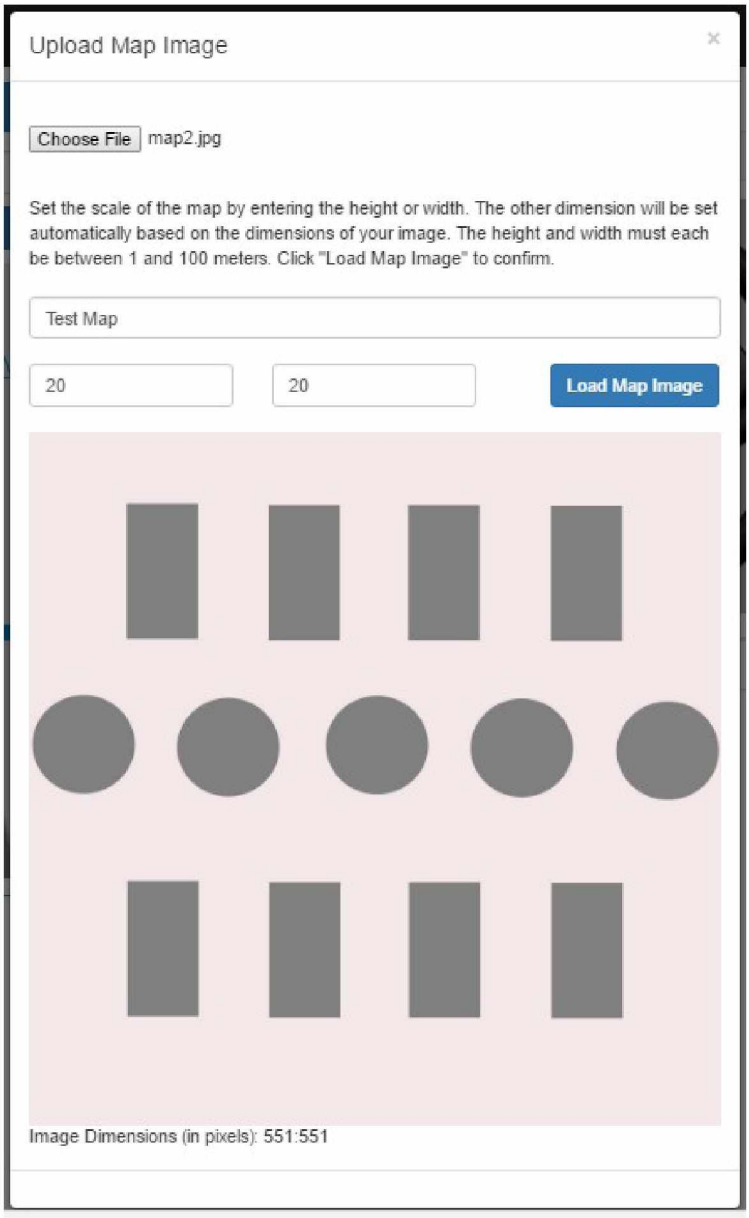

Figure 2: Upload Map Image Module. A preview of the map image is shown, and the user can set the scale of the map. After the user clicks "Load Map Image" the image is loaded into the Navigation module and 3D map.

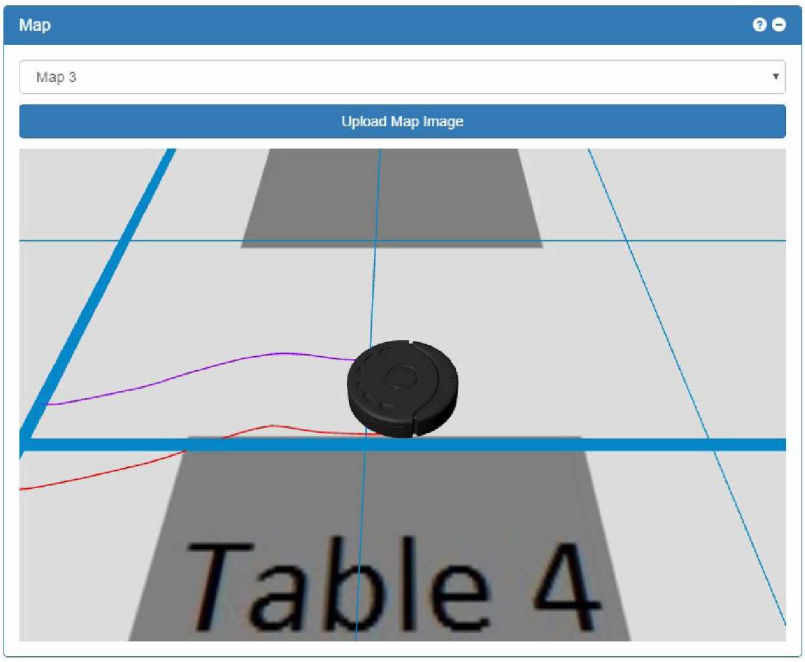

Figure 3: Uploaded image displayed in the 3D map.

A new module was added to the frontend user interface for navigation. The module displays the uploaded map image (or a blank plane if no image is uploaded) and robot location, which is updated in real time. The module interface offers three options for the user: The user can reset the robot location, add waypoints, or set obstacles.

To reset the robot location, the user sets the robot location on the map and sets the orientation. A command is sent to the robot to set its stored location coordinates and orientation to the new values.

If the user adds waypoints to the map, lines of JavaScript code are displayed to command the robot to drive in a path from waypoint to waypoint (see section 3.3 below for details on the navigation commands in the function library). The user is instructed to copy and paste the code into a state in the code editor. The user can then edit the commands or add more commands in the code editor. The user can run the code to send the driving commands to the robot. A map with waypoints is shown in Figure 4.

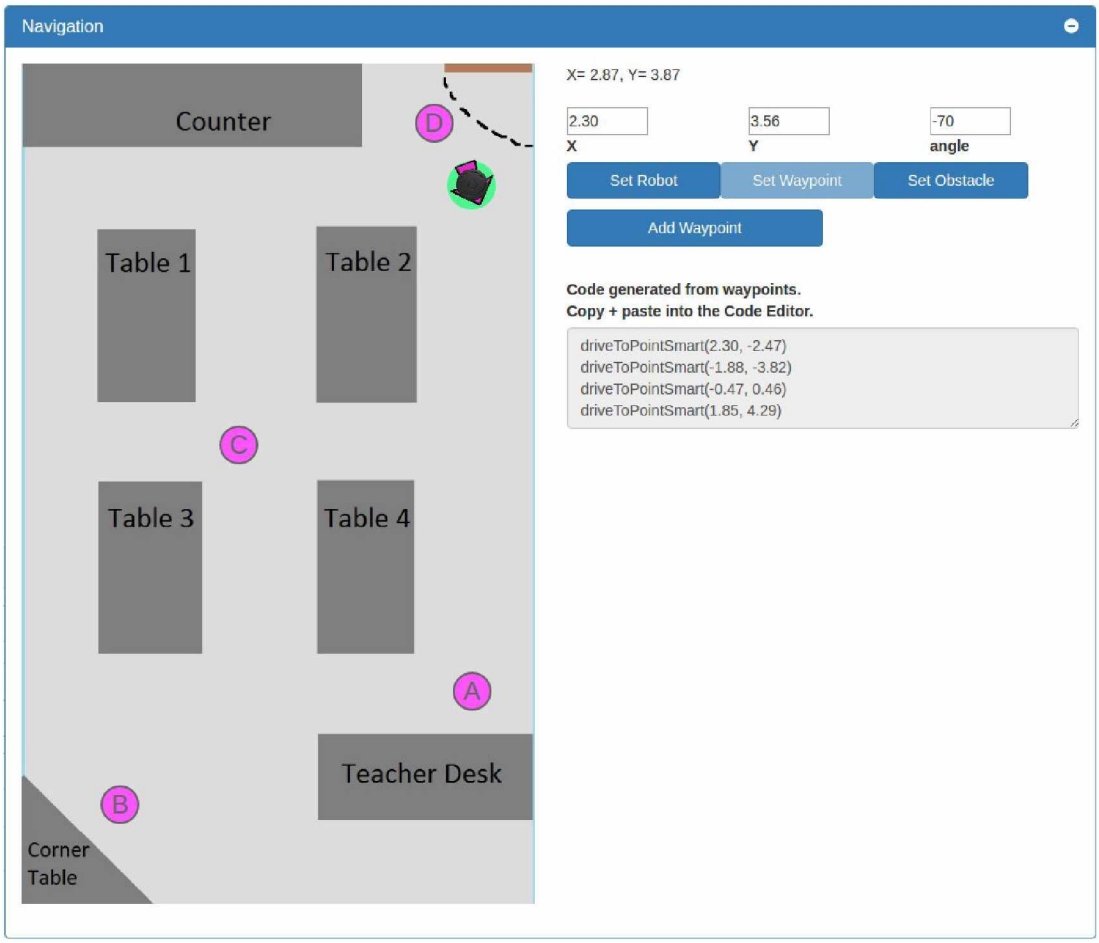

Figure 4: Navigation module with uploaded map image and four waypoints set. The robot location and orientation is represented by a robot image over a green circle and pink arrow

If the user adds obstacles to the map, the obstacles appear as red transparent squares over the map. The squares correspond to cells of an internal occupancy grid stored in memory that is used for navigation algorithms (described below in section 3.3). Cells corresponding to obstacles on the map are marked as occupied.

The user has the option to generate obstacles automatically from the map image. The algorithm looks at the map image at each cell and marks the cell as occupied if the average relative luminance [15] of pixels in the cell is above a given threshold. See Figure 5 for an example result.

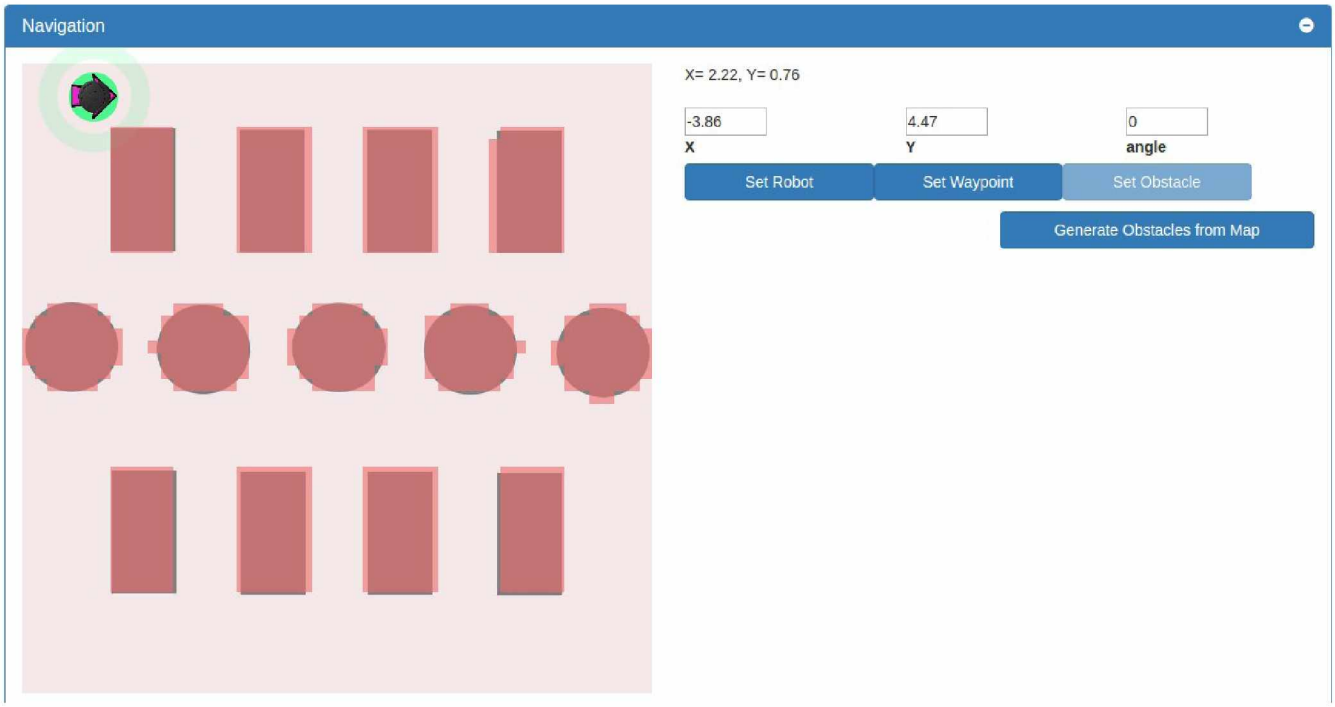

Figure 5: Result of generating obstacles automatically from a map image. Generated obstacles appear as transparent red squares.

The user can add or remove obstacles in real time while the robot is driving. The navigation algorithms are designed to update when the occupancy grid is changed, so if an autonomous navigation algorithm is running the user can watch the robot change its path and attempt to avoid obstacles while the user adds or removes them.

3.3 Function Library

Two functions were added to the RobotMoose code editor function library for autonomous navigation. The library already had basic drive functions to drive forward, backward, or turn for a given distance or angle, but it lacked functions for driving to a specific waypoint or finding a path through obstacles. The first added function commands the robot to drive directly to a given waypoint, and the second function expands on the first by using a path finding algorithm if the direct path is blocked.

The first function, driveToPoint(), takes the x and y Cartesian coordinates of a waypoint as input. It is a very simple function that uses the turn functions of the code library to turn the robot in the right direction, and then it uses the forward function to drive the robot until it reaches the given coordinates. The function does not check for obstacles, so the robot will continue to drive forward even if the path through the occupancy grid is blocked or if the robot runs into an object. This function was intended as a learning tool for users. After users have learned how to drive robots using the basic drive functions, they can use driveToPoint() to learn how to program the robot to drive using Cartesian coordinates, without worrying yet about obstacles or path finding.

The second function, driveToPointSmart(), also takes x and and y Cartesian coordinates as input. It first checks if the straight path between the current robot location and the given coordinates is clear of obstacles in the occupancy grid. As long as the path is clear, then the function works identically to driveToPoint(), except that it will repeatedly check that the path is still clear until the robot reaches the goal. If at any time it finds that the straight path is blocked by an obstacle, then it uses a modified A* algorithm with path smoothing to find a new path to the goal.

The algorithm used to check if the straight path is clear is a modified form of Bresenham's Algorithm [16]. Bresenham's Algorithm is frequently used in graphics to select cells in a grid to be used to approximate a straight line between two points. It can be adapted to find the grid cells intersected by a line between two points in the grid. The algorithm designates one coordinate axis as the *driving axis*, corresponding to the axis on which the distance between points is largest. It finds the cell containing the starting position and calculates an initial error bound ε, which is the negative distance between the point where the line exits the cell and the edge of the cell farthest from the starting point and parallel to the driving axis. That is, if the line from the start to end point is in the +x, +y direction, and the x

axis is the driving axis, then ε is the negative distance between the point where the line exits the cell and the "top" (+y) edge of the cell (see Figure 6, taken from the paper by Kenneth Joy cited above). The algorithm keeps track of the cell index i along the driving axis, and the index j along the other axis. The algorithm then proceeds to increment i by 1 and increment ε by the slope of the line m. If ε falls below zero, this means that the line has crossed into the next cell on the non-driving axis. In this case, j is incremented by 1, and ε is set to 1 - ε to become the new error bound of the next cell. To use Bresenham's Algorithm for checking obstacles, each cell selected in the algorithm is checked for obstacles in the occupancy grid.

Two modifications were made to Bresenham's Algorithm to use it for robot navigation. The unmodified algorithm is designed for drawing approximate lines with grid cells, so it misses some cells that the line passes through. These missing cells occur whenever the j index is incremented (i.e. when the line passes through the top of a cell), so the algorithm was easily modified to check this cell before incrementing the i index and moving on to the next cell.

The second modification is to account for the robot's width. If the cell sizes are small, or if the center of the robot passes near the edge of a cell, the robot may overlap into another cell when it is driving. The algorithm was modified to check additional cells on each side of the path – enough cells so that their total width is equal to or greater than half the width of the robot. This ensures that every cell that the body of the robot may intersect is checked.

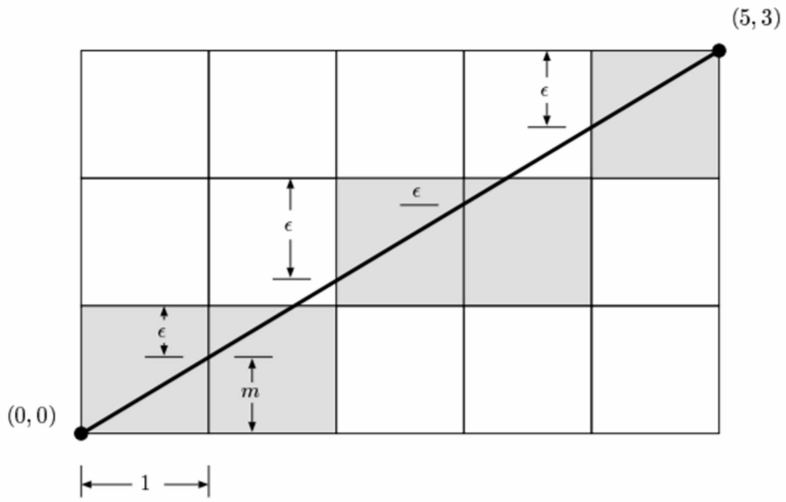

Figure 6: Illustration of Bresenham's algorithm.

The algorithm used for path finding when the path is blocked is the A* search algorithm [17]. A* finds a shortest distance path from one point to another in a graph, if a path exists. There may be more than one shortest path of the same distance, and A* will not find every shortest path, but it will find one of them.

A* is a modification of Dijkstra's algorithm. Dijkstra's algorithm expands the starting node by visiting each of its neighbors and calculating a tentative distance for that node, which is the path distance from the starting node to the expanded node plus the distance from the expanded node to the visited node. Then out of all visited nodes it expands the node with the shortest path distance from the starting node, and visits all of that node's neighbors that have not yet been expanded. Whenever a new tentative distance is calculated for a visited node, it replaces the old tentative distance if it is lower, and the new path to that node is recorded. It repeats this process until the goal node is visited, at which point a shortest path to the goal is known.

A* is the same as Dijkstra's algorithm, except instead of choosing which node to expand using only the distance of the path to the node, it uses the distance of the path plus some heuristic distance. The heuristic distance is application specific, but to guarantee that the

algorithm finds a shortest path the heuristic must be equal to or less than the shortest possible path between that node and the goal. Together the path distance and heuristic distance represent a lower bound for a path to the goal through this node. If nodes are expanded in order of lowest total distance, the first path found to the goal is guaranteed to be a shortest path.

The A* algorithm treats the robot as a single point, so a modification must be made to account for robot width. An algorithm was added as a preprocessing step before A* to add a layer of padding around the obstacles. The algorithm makes a copy of the occupancy grid, and around each obstacle it adds a layer of occupied cells to the copy equal to at least half of the robot's width. The A* algorithm uses this padded occupancy grid when it checks a grid cell. This ensures that it will not find a path where the robot body passes through a grid cell that is occupied in the original grid.

A* finds a shortest path along grid cells, but ground robots are not constrained to move along a grid. To take advantage of this, a smoothing algorithm is used to simplify and straighten the final path to the goal. For every pair of waypoints generated by the A* algorithm, the smoothing algorithm checks to see if the straight path between them is clear of obstacles. It uses the same modified Bresenham's Algorithm described above, except it ignores the second modification of the algorithm which accounts for the robot's width, since the A* algorithm already accounted for this when it padded the obstacles. If the straight path is clear, all intermediate waypoints between these two waypoints are removed from the path. When the code is run, the robot drives in a straight line between the two waypoints.

Whenever the occupancy grid changes while the driveToPointSmart() function is running and the robot is driving to the goal, the function checks again if the path to the next waypoint is still clear. If it is no longer clear, it runs the A* algorithm again to find a new path.

If the A* algorithm is unable to find a path, a command is sent to the robot to stop driving, and the function continues to run in a waiting mode. The function waits until the occupancy grid changes, at which point it tries again to find a path to the goal. It will continue running in this waiting mode indefinitely until a path is found.

Figure 7 shows an image of the driveToPointSmart( ) function running in debug mode with test output shown on the grid (normally not visible to the user). The output highlights grid cells in different colors to show the obstacle padding and path found by the A* algorithm.

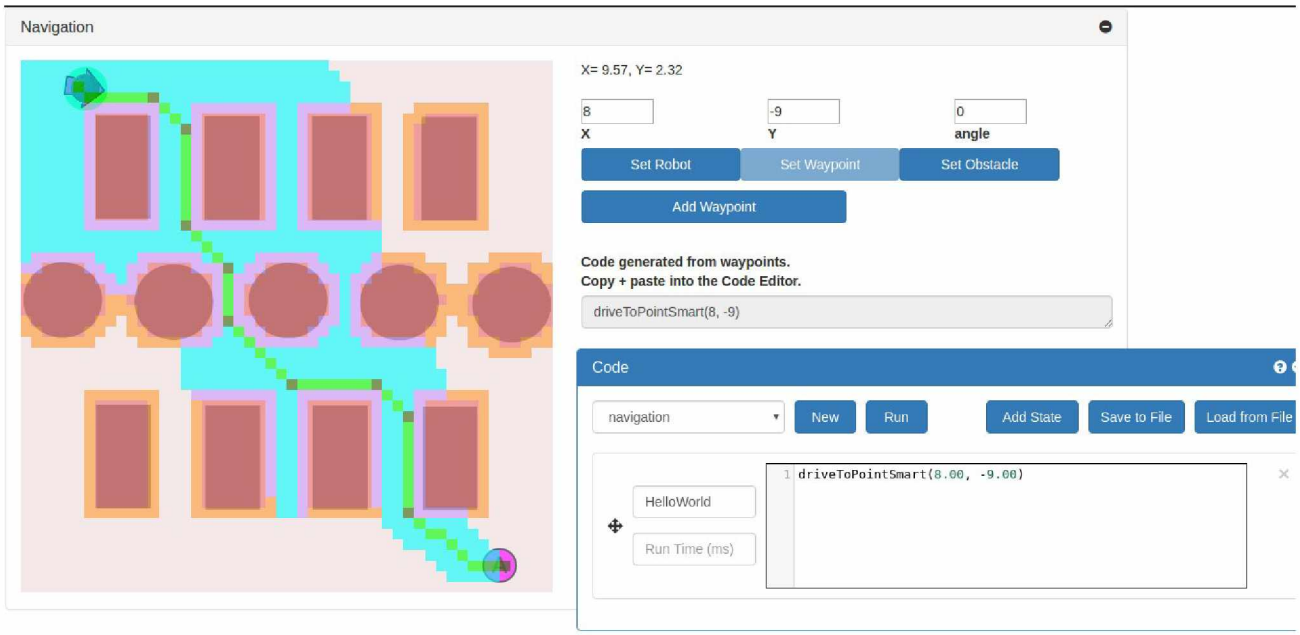

Figure 7: Running the driveToPointSmart( ) function, shown with output in debug mode. In the map image, red cells represent obstacles. Orange cells represent the layer of padding added around the obstacles for the A algorithm. Pink cells denote occupied cells checked by the A* algorithm. Light blue cells represent unoccupied cells checked by the algorithm. Light green cells represent the original path found by A*. Dark green cells denote the remaining points after path smoothing. When the function is run in the normal RobotMoose application the debug output is turned off, and only the red obstacle cells are displayed on the map.*

4 PERFORMANCE ANALYSIS

4.1 Benchmark Tests

The performance of the modified A* algorithm was tested for different grid cell sizes using one test map. The map was chosen to be 20 meters by 20 meters in size, with 15 large obstacles. For each grid size tested, obstacles were generated in the grid automatically from the map image. Since the same image was used for each grid size, the sizes of the obstacles were not scaled, so it was expected that the varying coarseness between cell sizes would introduce variance in the complexity of the path finding problem.

A set of 17 predetermined starting coordinates across the top of the map and 17 predetermined target coordinates in mirrored locations across the bottom of the map were used. The full, modified A* algorithm, including obstacle padding and path smoothing, was used to find a path between each pair of starting and target coordinates, for a total of 289 unique paths. Each trial was timed, and the average time over all the trials was recorded. The map used and one pair of starting and ending coordinates used in the tests can be seen in Figure 7 above.

The tests were run in a Google Chrome Web Browser. The test scripts were called from a function in the RobotMoose frontend Code Editor (in the same way that the navigation functions are called), with the RobotMoose application served from a local host server.

4.1 Results

The results of the performance benchmark tests are shown in Table 1.

Cell Width (m)	Number of Grid Cells	Average Run Time (ms)	Time Per Cell (ns)
0.4	2,500	1.1	440
0.2	10,000	4.6	458
0.1	40,000	25.9	647
0.05	160,000	150.9	943
0.025	640,000	736.4	1151

Table 1: A Performance on 20 x 20 meter Grid.*

The results show that the average time per run starts at about 1 millisecond for a very coarse grid with cell width of 0.4 meters. The average time increases in proportion to the number of grid cells at a rate moderately faster than linear. For each step in cell width, the number of grid cells increases by a factor of 4, while the average run time increases by a factor of between about 4.2 and 5.8.

The faster than linear growth in time may be explained by several factors. One factor is the addition to A* of the algorithm to pad obstacles. When the cell width is smaller, the algorithm has to add more cells on each side of an obstacle to accommodate the width of the robot, so the run time of this algorithm grows due to increases in both grid size and padding size.

Another factor is that at a certain recursion depth the algorithm pauses (using the Javascript setTimeout function with a wait of 0 milliseconds) to prevent web browser recursion limit errors and to prevent the GUI from locking up. This recursion depth may never be reached for coarse grid sizes or small maps. On the map used for these tests, some

experimentation indicated that the recursion depth was only reached for cell sizes of 0.1 m or smaller, which could explain the faster growth rate in run time at these sizes.

The algorithm was successful in finding a path in every test, and the average run time seems reasonable for real-time navigation using cell sizes as small as 0.1 m. For grid cells smaller than 0.1 m, the run times are long enough to introduce observable delays in the user interface.

Table 2 shows the memory usage of the A* algorithm during the tests for some grid sizes. The algorithm allocated between approximately 26-62 bytes per grid cell, depending on the grid size.

Number of Grid Cells	Total RAM (KB)	Bytes Per Cell
2,500	154	62
10,000	497	50
40,000	1,022	26
160,000	6,133	38

Table 2: A Memory Usage.*

5 CONCLUSION AND FUTURE WORK

The project appears to achieve its goal of expanding RobotMoose to enable a user to upload a map, plan and test a path through the map, and program the robot to navigate autonomously through this path. The navigation algorithms were able to find a path in every configuration tested, and they are efficient enough to be used in real time with changing obstacles, up to a certain limit of grid size.

The performance of the modified A* algorithm was not fast enough for real time user interactivity for grid cells smaller than 0.1 m, so finding methods of improving its efficiency could be helpful. Identifying large obstacles or large portions of the graph clear of obstacles could be one way of improving the scaling. This could be achieved by using adaptive grid cell sizes, by using separate layers of grids of different cell sizes, or by checking the line of sight throughout the algorithm rather than individual cells.

The algorithm performed reasonably well with grid cells of 0.1 m or larger in a 20 m x 20 m space. So for spaces with known maps where the robot does not need to navigate through close obstacles, this performance may be sufficient for real-time navigation.

The functionality provided by the user interface and navigation algorithms of this project could allow a user to become familiar with robot programming and autonomous navigation. The drive functions simplify navigation while still giving the user experience with programming.

Together with the pre-existing RobotMoose tools, the new tools enable the user to program more complicated tasks. RobotMoose already allowed a user to operate and program actuators such as servo motors, audio speakers, and LEDs, and it allowed the user to programmatically add custom buttons or display text on the frontend. Now these tools can be used alongside the new user interface module and library functions for robot navigation. For

example, a user could program a telepresence robot to lead another remote user on a guided tour through a space by driving between set waypoints. At each waypoint in the tour, the robot could be programmed to display text, play a sound, ask the user on-screen questions, or perform other actions.

There are a number of ways that the RobotMoose tools could be further expanded. First of all, localization and mapping algorithms could be added either to the frontend or backend to use sensor data to update the occupancy grid in real time. These algorithms are often complex and specific to the sensors available on the robot, but it would not be difficult to integrate them with the autonomous navigation functions of this project. The algorithm used in this project already accounts for updates in the occupancy grid, so the navigation functions would not need to be changed. The mapping algorithms would simply need to update the occupancy grid used by the navigation functions.

Other tools could be added to RobotMoose to expand on the navigation functionality and increase the ways the robot can interact with humans or the environment. For example, a function could be added to allow the robot to follow another robot, human, or object. It would be simple to adapt the navigation functions to navigate toward a moving waypoint rather than a static waypoint, or the object to follow could be marked uniquely on the occupancy grid. Backend software could track the object using a variety of different methods, such as following a QR code.

Other tools that could be added to enable more interaction with humans include facial recognition software, voice commands, or text-to-speech software to enable a user to program a robot to give audio messages.

One limitation of the navigation functions is that they are implemented in the RobotMoose code editor as blocking functions. They do not block the whole RobotMoose application, but they block most other functions from running at the same time in the code

editor. This was a decision made to simplify programming in the code editor by matching the way the pre-existing functions work, but it means that there is no simple way to programmatically send other commands to the robot while it is driving to a waypoint. The navigation functions or the entire code editor system could be reworked to allow for concurrent robot commands. This would allow freedom for decisions regarding interactions with humans or the environment while the robot is navigating, but it would need to be implemented carefully to preserve the simplicity and accessibility of the user interface.

REFERENCES

[1] Stefan Schaal, "The new robotics—towards human-centered machines." HFSP Journal, 1:2, pp. 115-126, 2007. http://dx.doi.org/10.2976/1.2748612 (accessed April 2, 2017).

[2] S. Šabanović, M.P. Michalowski, R. Simmons. "Robots in the Wild: Observing Human-Robot Social Interaction Outside the Lab" *Proceedings of the IEEE International Workshop on Advanced Motion Control (AMC 2006),* Istanbul Turkey, March 2006. http://ieeexplore.ieee.org/document/1631758/ (accessed April 17, 2017).

[3] Andrea Gomolli, Cindy E. Hmelo-Silver, Selma Šabanović, Matthew Francisco. "Dragons, Ladybugs, and Softballs: Girls' STEM Engagement with Human-Centered Robotics." J Sci Educ Technol 25: 899, 2017. https://link.springer.com/article/10.1007%2Fs10956-016-9647-z (accessed April 17, 2017).

[4] RobotMoose. https://robotmoose.com/ (accessed April 2, 2017).

[5] "iRobot Create 2 Programmable Robot," iRobot. http://www.irobot.com/About-iRobot/STEM/Create-2.aspx (accessed April 2, 2017).

[6] "RobotMoose," GitHub. https://github.com/robotmoose/robotmoose/ (accessed April 2, 2017).

[7] ROS.org. http://wiki.ros.org/ (accessed April 2, 2017).

[8] "Navigation," ROS.org. http://wiki.ros.org/navigation (accessed April 2, 2017).

[9] "Rviz," ROS.org. http://wiki.ros.org/rviz (accessed April 2, 2017).

[10] "Wviz," ROS.org. http://wiki.ros.org/wviz (accessed April 2, 2017).

[11] V-Rep: Virtual Robot Experimentation Platform. http://www.coppeliarobotics.com/ (accessed April 2, 2017).

[12] "Occulus Prime Mobile Robots," Xaxxon. http://www.xaxxon.com/ (accessed April 2, 2017).

[13] "InTouch Health and iRobot to Unveil the RP-VITA™ Telemedicine Robot at Clinical Innovations Forum." InTouch Health. http://www.intouchhealth.com/media/press-release/07-24-2012/ (accessed April 17, 2017).

[14] Tickle. https://tickleapp.com/ (accessed April 2, 2017).

[15] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta, "A Standard Default Color Space for the Internet – sRGB," W3C. https://www.w3.org/Graphics/Color/sRGB (accessed April 2, 2017).

[16] Kenneth I. Joy, "On-Line Computer Graphics Notes: Bresenham's Algorithm." UC Davis. http://graphics.idav.ucdavis.edu/education/GraphicsNotes/Bresenhams-Algorithm.pdf (accessed April 2, 2017).

[17] Peter E. Hart, Nils J. Nilsson, Bertram Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” IEEE Transactions on Systems Science and Cybernetics, 4:2, 1968. http://ieeexplore.ieee.org/document/4082128/ (accessed April 2, 2017).